На прошлом занятии мы с вами получили класс для представления времени:

```language-cpp
class TimeDay {
    unsigned long time {0};

public:
    TimeDay() : time(0)
        { }
    TimeDay(unsigned char hs, unsigned char ms, unsigned char ss, unsigned int
ds = 0)
        : time(ss + ms * 60 + hs * 3600 + ds * 86400)
        { }
    TimeDay(unsigned long tm)
        : time(tm)
        { }

    void get_time(unsigned int& days, unsigned char& hours, unsigned char& mins,
unsigned char& secs) const
    {
        secs = time % 60;
        mins = time/60 % 60;
        hours = time/3600 % 24;
        days = time/86400;
    }

    unsigned long get_time() const { return time; }
};
```

И определили операцию сложения для его объектов с помощью функции:

```language-cpp
unsigned long operator + (const TimeDay& left, const TimeDay& right)
{
    return left.get_time() + right.get_time();
}
```

Как видим, эта функция обращается к методу get_time переданных объектов – операндов для сложения, чтобы получить доступ к приватному значению поля time. **И было бы логично расширить на нее область видимости класса TimeDay, чтобы в теле этой функции можно было бы напрямую обращаться к приватным данным объектов этого класса**. Язык C++ предоставляет такой легальный механизм обхода ограничения для доступа к приватным атрибутам вне класса. Делается это путем объявления функции operator+ дружественной по отношению к классу TimeDay:

```cpp
class TimeDay {
    unsigned long time {0};

public:
...
    friend unsigned long operator + (const TimeDay& left, const TimeDay& right);
};
```

То есть, **пишется ключевое слово friend, за которым следует прототип дружественной функции. После этого в теле этой функции становится возможным прямое обращение к полю time объектов left и right**:

```cpp
unsigned long operator + (const TimeDay& left, const TimeDay& right)
{
    return left.time + right.time;
}
```

Надо сказать, что концепция дружественных функций весьма спорная в языке C++. Одни считают ее приемлемой, т.к. редко, но существует необходимость распространить область видимости класса на внешние функции. Другие полагают, что это вносит больше путаницы в текст программы и, как следствие, к появлению неочевидных проблем, т.к. дружественные функции могут менять состояние объектов, напрямую через приватные переменные, а не через публичные методы, как это предусматривает разработчик класса. Тем не менее, этот инструмент существует в языке C++, который, как минимум, нужно знать. **Однако применять на практике следует с очень большой осторожностью и несколько раз подумать, а можно ли сделать то же самое иначе без неоправданного расширения области видимости класса**.

Дружественные классы

Помимо дружественных функций можно объявлять и дружественные классы. Делается это по аналогии. Я приведу учебный, несколько искусственный пример такого дружественного класса. В действительности, необходимость объявлять целый класс другом другого класса – крайне редкая ситуация. И возникает она, как правило, в нетривиальных проектах. Поэтому здесь будет лишь простая демонстрация этой возможности.

В классе TimeDay объявим дружественным класс Clock (часы):

```cpp
class TimeDay {
    unsigned long time {0};
```

```
    friend class Clock;

public:
...
};
```

Обратите внимание, что дружественность можно прописывать в любых секциях (private, public), разницы никакой нет.

После класса TimeDay объявим класс Clock, например, так:

```language-cpp
class Clock {
public:
    void show_time(const TimeDay& t)
    {
        unsigned char secs = t.time % 60;
        unsigned char mins = t.time/60 % 60;
        unsigned char hours = t.time/3600 % 24;

        printf("%02u:%02u:%02u", hours, mins, secs);
    }
};
```

И, затем воспользуемся им в функции main:

```language-cpp
int main()
{
    TimeDay t1(10, 45, 13), t2(4, 11, 50);
    TimeDay res = t1 + 10;

    Clock cl;
    cl.show_time(res);

    return 0;
}
```

Благодаря дружественности с классом TimeDay в методе show_time класса Clock мы имеем возможность напрямую обращаться к приватной переменной time объекта t.

Конечно, тот же самый функционал легко можно было бы сделать без всякой дружественности и это было бы правильнее. Это исключительно показательный, учебный пример, не более того. Вообще, на практике, если вы задумались об использовании дружественной функции или, тем более, класса, то подумайте еще раз, возможно, несколько стоит изменить

структуру программы, отредактировать взаимосвязи между классами и сами классы. И если только после этого видите, что использование дружбы оправданно, пишите ключевое слово friend, не раньше!